CS325: Analysis of Algorithms, Fall 2020

Practice Assignment 2 Solution

Problem 1.

Algorithm Description: Let $p[i].x$ and $p[i].y$ denote X and Y coordinates of $p[i]$. The algorithm outputs S, the set of all maximal points. Initially, S is empty.

(a) Sort all points based on their X coordinates. Let $p[1], p[2], ..., p[n]$ be the order of the points obtained, that is $p[1].x \leq p[2].x \leq ... \leq p[n].x$

(b) Add $p[n]$ to the set S. Let cur be the index of the last point added to S by the algorithm. Set, $cur = n$. Note that, this point has the largest Y-coordinate among all maximal points in S.

(c) The algorithm considers all $p[i]$'s iteratively, from $p[n-1]$ to $p[1]$.

(d) When $p[i]$ is considered, if its Y-coordinate is larger than cur, the algorithm adds it to S and updates cur to i, otherwise, the algorithm disregards $p[i]$.

Can you come up with a recursive formulation of this iterative algorithm? (it is simpler)

Pseudocode: Let $p[i].x$ and $p[i].y$ denote the value of x and y coordinate of the ith point, where $i = 1, 2, ..., n$.

```
STAIRCASE
    sort list p by their x coordinate value. (use merge sort)
    S ← ∅ as empty
    S.insert(p[n])
    cur ← n
    for i = n − 1 to 1
        if p[i].y ≥ p[cur].y
            S.insert(p[i])
            cur ← i
        end if
    end for
return S
```

Proof. We prove that after the ith iteration of the for loop, S contains the maximal points of $\{p[n-i], p[n-i+1], \ldots, p[n]\}$, which, in particular, implies that S contains all maximal points in the end. We use induction on i. The **base case** is for $i = 0$. S contains $p[n]$ the only maximal point of $\{p[n]\}$ before any iteration of the for loop.

Induction Hypothesis ensures that for any $j < i$ we have the desired property, that is after the jth iteration S contains maximal points of $\{p[n-j], p[n-j+1], \ldots, p[n]\}$.

Induction step is to prove the statement for i, that is after the ith iteration S contains maximal points of $\{p[n-i], p[n-i+1], \ldots, p[n]\}$. By induction hypothesis, after $i-1$ iterations S contains maximal points of $\{p[n-i+1], p[n-i+2], \ldots, p[n]\}$. Note that these are also maximal points of $\{p[n-i], p[n-i+1], \ldots, p[n]\}$, as $p[]$ is sorted by X-coordinates. Also, $p[i]$ is a maximal point if and only if its Y-coordinate is larger than all maximal points of $\{p[n-i+1], p[n-i+2], \ldots, p[n]\}$ (Why?). This condition is checked by the algorithm. $\square$

Running time: The algorithm spends $O(n \log n)$ time to sort the points (merge sort). After that, it spends $O(1)$ time per iteration. So the total running time is $O(n + n \log n)$, which is $O(n \log n)$.

Problem 2. A sequence $X[1, \ldots, n]$ is bitonic if $X[1, \ldots, i]$ is increasing and $X[i, \ldots, n]$ is decreasing for some i with $1 < i < n$. If we can find this value of i, then we can perform binary searches for k on the sequences $X[1, \ldots, i]$ and $X[i, \ldots, n]$ as they are both sorted. Binary search on these sequences will take $O(\log i) \in O(\log n)$ and $O(\log(n-i+1)) \in O(\log n)$ respectively, so if we can find the value i in $O(\log n)$ time, our entire algorithm will take $O(\log n)$ time. We first make some observations about bitonic sequences.

Observation 1 We can determine whether an index j is less than i, equals i, or is greater than i by comparing $X[j]$ to $X[j-1]$ and $X[j+1]$. If $j > i$, $X[j-1] > X[j] > X[j+1]$ as $X[i, n]$ is decreasing. If $j < i$, $X[j-1] < X[j] < X[j+1]$ as $X[1, \ldots, i]$ is increasing. If $j = i$, $X[j-1] < X[j] > X[j+1]$.

Observation 2 For any $j \leq i \leq k$, the sequence $X[j, \ldots, k]$ is bitonic. Indeed, the sequence $X[j, i]$ is increasing because it is a subsequence of the increasing sequence $X[1, \ldots, i]$. Likewise, the sequence $X[i, k]$ is decreasing.

Algorithm We are now ready to give an algorithm for finding i. Our algorithm is a recursive algorithm similar to binary search. If $n = 1$, then we return 1. Otherwise, let $\texttt{mid} = \lfloor n/2 \rfloor$. We compare i to `mid` using Observation 1. If $i = \texttt{mid}$, we return `mid`. If $i < \texttt{mid}$, then we recurse on $X[1, \ldots, \texttt{mid}-1]$ and return the output of the recursion. If $i > \texttt{mid}$, then we recurse on $X[\texttt{mid}+1, n]$. The indices of elements in $X[\texttt{mid}+1, n]$ are not the same as the indices of the same elements in $X[1, \ldots, n]$. (What is the index of 3 in [1,2,3,4]? What is the index of 3 in [3,4]?) Accordingly, we return the output of the recursion + `mid+1`.

Proof of Correctness We prove the correctness of algorithm by induction of the size of the sequence n. For $n = 1$, we return the only possible value for i: 1. Now suppose our algorithm works for all values of k for $1 \leq k \leq n-1$. Consider a bitonic sequence $X[1, \ldots, n]$. If $\texttt{mid} = i$, then our algorithm will correctly return i. If $i < \texttt{mid}$, then by Observation 2, the sequence $X[1, \ldots, \texttt{mid}]$ is bitonic. By our inductive hypothesis, our algorithm will correctly return i for $X[1, \ldots, \texttt{mid}]$ as this sequence has $\texttt{mid} = \lfloor n/2 \rfloor \leq n-1$ elements. The same argument applies $X[\texttt{mid}, \ldots, n]$, except we have to take extra care to return the correct index.

Running Time Analysis Our algorithm has the recursive running time $T(n) = T(n/2) + O(1)$. From the runnning time analysis of binary search, we know this recursive relationship solves to $T(n) = O(\log n)$

Problem 3. We will design a dynamic program to determine if the subsequence $A[i, \ldots, j]$ is oscillating for all $1 \leq i < j \leq n$. Specifically, we will fill a boolean dynamic programming table D with $D[i, j]$ storing the truth value of whether $A[i, \ldots, j]$ is oscillating. We will then show how we can use this table D to find the longest oscillating subsequence of X.

We begin with a few observations about oscillating sequences.

Observation 1 Let $A[i, \ldots, j]$ be an oscillating subsequence. For any $i \leq k \leq j$, the subsequence $X[i, \ldots, k]$ is also oscillating.

Observation 2 Consider the subsequence $A[i, \ldots, j]$. The index of the element $A[j]$ in $A[i, \ldots, j]$ is $j - (i - 1)$.

Observation 3 Let $1 \leq i < j \leq n$. The values of the dynamic programming table have the follwing recursive relationship:

$$D[i, j] = \begin{cases} D[i, j-1] \,\&\&\, A[j-1] < A[j] & \text{if } j - i \text{ even} \\ D[i, j-1] \,\&\&\, A[j-1] > A[j] & \text{if } j - i \text{ odd.} \end{cases}$$

We now prove the above relationship. Assume $A[i, \ldots, j]$ is oscillating. By Observation 1, $A[i, \ldots, j-1]$ is oscillating. If $j - i$ is even, then the index of $j - 1$ in $A[i, \ldots, j]$, $(j - 1) - (i - 1)$, is even, so $A[j-1] < A[j]$. If $j-i$ is odd, then $A[j-1] > A[j]$. Now suppose that $A[i, \ldots, j-1]$ is oscillating, $j-i$ is even, and $A[j-1] < A[j]$. We claim that $A[i, \ldots, j]$ is oscillating. We need to verify that the definition of oscillating sequence holds for the odd and even indices of $A[i, \ldots, j]$. The odd indices of $A[i, \ldots, j]$ are $k = i, \ldots, j$. It is true that $A[k] > A[k + 1]$ for $k = i, \ldots, j - 2$ as $A[i, \ldots, j]$ is an oscillating sequence. This condition is vacuously true for $k = j$ as there is no $A[j + 2]$ to compare to $A[j]$. The even indices of $A[i, \ldots, j]$ are $k = i + 1, \ldots, j - 1$. It is true that $A[k] < A[k + 1]$ for $k = i+1, \ldots, j-3$ as $A[i, \ldots, j-1]$ is oscillating. It is also true that $A[j-1] < A[j]$ by assumption. Therefore, $A[i, \ldots, j]$ is oscillating. The case when $j - i$ is odd is similar.

Algorithm We will iterate through the subsequences of A by their length k and their first element i. Initially, all length 1 subsequences $D[i, i]$ are set to true. For $k > 1$, we use the recursive formula in Observation 3 to compute $D[i, i+k]$. When we compute $D[i, i+k]$, the table entry $D[i, i+(k-1)]$ will have already been computed.

To find the longest oscillating subsequence, we iterate through the subsequences in reverse order of length k and first element i and return the first k such that $D[i, i + k]$ is true.

Proof of Correctness The correctness of this algorithm follows from the correctness of Observation 3.

Running Time Analysis There are $O(n^2)$ elements $D[i, j]$ of D, so it takes $O(n^2)$ time to compute D as computing an element $D[i, j]$ takes constant time. Finding the largest subsequence using D involves iterating through D, which also takes $O(n^2)$ time. Our algorithm takes $O(n^2)$ time in total.